Projet d'Imagerie Polytech'Nice Sophia Antipolis - Option TNS

A. Fournier sous la direction de X.Descombes

13/09/07

Introduction

Ce document se présente comme un guide de survie pour les élèves de l'option TNS suivant le projet d'imagerie. Il contient d'abord une brève présentation des sujets à traiter suivi d'un tutorial sur la bibliothèque de traitement d'images CImg. Quelques notions essentielles sur le langages C++ seront abordées avant d'évoquer quelques conseils généraux

Table des matières

1 Présentation des sujets

Chaque groupe devra choisir un article parmi les quatre proposés. Chaque article présente une méthode ou une série de méthodes permettant de résoudre un problème classique de traitement d'images. Le travail proposé est l'implantation informatique de la méthode proposée et l'application sur des images de test.

1.1 Jetstream

Il s'agit d'une méthode de détection de contours basée sur du *filtrage particulaire*. Le filtrage particulaire peut-être vu comme une évolution du filtre de Kalman discretisé. En pratique, celà consiste à générer plusieurs particules pour suivre un contour. On fait évoluer les particules sur l'image selon certaines régles et on sélectionne celles qui sont le plus représentatives du modèle *a priori* de contour pour générer les particules à l'étape suivante.

1.2 Discrimination de textures

Cet article présente une méthode de segmentation d'images texturées par « filtrage optimal ». L'intérêt de cette approche est que ce sont des techniques assez générales de traitement de signal qui sont utilisées plutôt que des modèles spécifiques au traitement d'image. La méthode permet de discriminer deux textures différentes sur la même image. Le travail demandé consistera à calculer et élaborer le filtre optimal ainsi que d'analyser les résultats.

1.3 Filtrage numérique par variation totale

Les méthodes variationnelles font partie des méthodes classiques en traitement d'images. Il s'agit souvent de trouver et minimiser une fonctionnelle sur une image ce qui se traduira comme une ou plusieurs équations aux dérivées partielles. Le filtre TV peut être utilisé comme une méthode de restauration d'images bruitées, métant en œuvre une version numérique des filtres fondamentaux en traitement d'image par méthodes variationnelles. Il faudra implanter l'algorithme de Variation Totale et comparer les résultats avec les filtres classiques sur des images Noir et Blanc ainsi que des images couleurs.

1.4 Détection de Flamants Roses par Processus Ponctuels Marqués pour l'estimation de la taille des populations

Les processus ponctuels sont champ assez récent en traitement d'images. Il s'agit de processus aléatoires permettant la naissance ou la mort (apparition, disparition) d'objets selon certaines conditions. Ces conditions vont se traduire par un modèle *a piori* sur les intéractions entre les objets et un modèle d'attache aux données (évaluation de l'adéquation de l'objet avec l'image). Les propriétés des objets dépendent de l'application : segments alignés pour l'extraction de réseaux routiers, polygones pour la reconstitution d'un Modèle Numérique d'Élévation... Dans cet application, on utilisera des cercles afin de détecter des Flamands roses sur des prises de vue aériennes

2 Langage d'implémentation

Vous implanterez le programme en C++. Vous travaillerez avec la bibliothèque graphique CImg, qui met en œuvre de façon aisée, les fonctions fondamentales du traitement d'image (acquisition, affichage, et diverses manipulations).

http ://cimg.sourceforge.net/

Attention : CImg seule ne sait lire que des images au format ppm. Pour lire (presque) tous les types d'images avec CImg, vous devrez également installer Image Magick :

http ://www.imagemagick.org/script/binary-releases.php

La documentation de CImg est particulièrement fournie et explicite. néanmoins, pour commencer, voici les concepts fondamentaux à acquérir pour une utilisation basique :

La classe de référence qui manipule les images est la classe template CImg :

```
CImg<T>
//déclare une image dont les pixels seront représentés
//par des objets de type T (en général des unsigned char
//pour des entiers sur 8 bits, mais on peut prendre
//des flottants ou des entiers si c'est nécessaire)
```

Parmi les constructeurs de cette classe, on remarquera :

```
CImg<T> monImage(int dx,int dy,int dz,int c);
//permet de créer une image de taille dx par dy par dz
et possédant c canaux (pour les couleurs, entre autres).
```

```
Cimg<T> monImage(const char * nom_de_fichier);
//permet de créer une image à partir d'un fichier
```

Enfin, LA méthode fondamentale de cette classe :

```
T& operator(int colonne, int ligne);
//pour accéder aux pixels en lecture et écriture:
monImage(5,5);
//accède au pixel de la sixième ligne et de la sixième
//colonne
```

Si vous utilisez CImg, regardez la documentation avant de vous attaquer à une opération sur l'image, certaines fonctions déjà présentes dans la bibliothèque vous éviteront d'avoir à réinventer la roue.

3 Le traitement d'images, le C++ et CImg

Le C++ est un langage puissant, orienté objet et héritant de la syntaxe et des principes du C. Normalement, une utilisation à la C pourrait fonctionner parfaitement, mais certaines notions pourraient vous poser quelques problèmes.

3.1 Constructeur, constructeur par copie et constructeur par défaut

En C++ (comme en Java), les objets (instances de class ou struct) possèdent une ou plusieurs fonctions membres appelées « constructeur ». Le constructeur sert à initialiser les attributs de l'objet. Si dans une fonction, vous déclarez un objet autre que char, int, double ... (types de bases), alors, cet objet sera non seulement déclaré mais également initialisé (*i.e.* construit). La syntaxe de déclaration d'un objet dans une fonction est donc :

```
void ma_fonction(double mon_nombre)
{
  ...
  //déclaration et initialisation d'un objet:
  CImg<int> mon_image("mon_image.jpg");
  //mon_image est une instance de la classe CImg<int>,
  //ici, on invoque le constructeur qui prend comme
  //paramètre un char* donnant le nom du fichier
  //où lire l'image
  ...
}
```

Attention, toutefois. Beaucoup d'objets possèdent un constructeur « par défaut ». C'est à dire un constructeur ne demandant pas de paramètres. C'est lui qui sera invoqué par la syntaxe suivante :

```
void ma_fonction(double mon_nombre)
{
  ...
  //déclaration et initialisation d'un objet:
  CImg<int> mon_image;
  //ATTENTION, ici le constructeur par défaut est appelé.
  //À vous de regarder dans la documentation de CImg
  //pour connaître son comportement (quelle taille a
  //l'image créée ?)
  ...
}
```

Il existe aussi un constructeur par copie, qui vous permet d'initialiser un objet en copiant les données d'un autre objet (souvent de même type) :

```
void ma_fonction(double mon_nombre)
{
  ...
  //déclaration et initialisation d'un objet:
  CImg<int> mon_image1("mon_image.jpg");
  //appel au constructeur qui prend comme paramètre
  //un char*
  CImg<int> mon_image2(mon_image1);
  //appel au constructeur par copie, image2 sera la copie
  //de image1
  ...
}
```

La syntaxe suivante est équivalente :

```
void ma_fonction(double mon_nombre)
{
  ...
  CImg<int> mon_image1("mon_image.jpg");
  //appel au constructeur qui prend comme paramètre
  //un char*
  CImg<int> mon_image2=mon_image1;
  //appel au constructeur par copie, image2 sera la copie
  //de image1
  ...
}
```

Attention le code suivant a un comportement différent :

```
void ma_fonction(double mon_nombre)
{
  ...
  CImg<int> mon_image1("mon_image.jpg");
  //appel au constructeur qui prend comme paramètre
  //un char*
  CImg<int> mon_image2;
  //appel au constructeur par défaut
  mon_image2=mon_image1;
  //appel à l'operateur d'affectation
  ...
}
```

Comme vous pouvez le voir, ici image2 est construite via le constructeur par défaut puis on utilise l'operateur d'affectation (l'opérateur =) pour lui affecter la valeur de image1. Pas de panique, l'opérateur = et le constructeur de copie sont la plupart du temps équivalents, mais rappelez-vous que vous appelez deux fonctions différentes.

3.2 Passage par référence et passage par valeur

Supposons que pour les besoins de votre algorithme vous ayez fait une fonction auxiliaire qui modifie votre image. Si vous êtes habitués à travailler en C, alors vous écrirez probablement quelque chose de ce genre :

```
CImg<int> fonction_auxiliaire(CImg<int> image_a_traiter)
{
  ...//modifications de image_a_traiter
  return image_a_traiter;
}
```

Vous appelerez cette fonction probablement de cette façon :

```
mon_image_modifiee=image_a_traiter(mon_image)
mon_image=mon_image_modifiee;
```

Où plus simplement :

```
mon_image=image_a_traiter(mon_image)
```

Ceci fonctionne, mais peut devenir *très, très lent*. Pourquoi ? Parce qu'image_a_traiter sera une copie locale de mon_image dans fonction_auxiliaire. C'est à dire que le constructeur de copie de la classe CImg<int> sera utilisé pour créer image_a_traiter à partir de mon_image. Autrement dit, l'appel de la fonction sera équivalent au code suivant :

```
CImg<int> image_a_traiter=mon_image.
...//modifications de image_a_traiter
Cimg<int> retour_de_fonction_auxiliaire=image_a_traiter;
mon_image=retour_de_fonction_auxiliaire;
mon_image=mon_image_modifiee;
```

Ainsi, toute l'image sera copiée pixel par pixel à trois reprises, et ce inutilement à chaque fois que vous appelez fonction_auxiliaire. Si l'image est un peu grande et que cette fonction est bien placée dans une bonne grosse boucle *for*, vous pouvez être sûrs que vous allez mettre votre processeur à genoux.

Pour rémédier à ça, le C utilise la notion de pointeur, mais la syntaxe qui en découle est dangereuse et peu aisée à utiliser. Le C++ propose le mécanisme de passage de paramètre *par référence*. Passer un paramètre *par référence* permet de travailler véritablement sur l'objet passé et ainsi éviter les copies. La syntaxe est très simple. On peut réécrire fonction_auxiliaire de la façon suivante :

```
void fonction_auxiliaire(CImg<int>& image_a_traiter)
//notez l'ajout du "&"
{
  ...//modifications de image_a_traiter
}
```

Deux choses à noter

- On ajoute un « & » après le type d'entrée pour indiquer qu'on passe l'image *par référence*.
- fonction_auxiliaire ne renvoie plus rien : plus besoin puisqu'elle travaille directement sur l'image qu'on lui passe en paramètre !

Pour utiliser la fonction, rien de plus simple :

```
fontion_auxiliaire(mon_image);
```

Ainsi, mon_image sera modifiée par fonction_auxiliaire. Cet appel sera alors équivalent à :

```
...//modifications de mon_image
```

3.3 Astuces du C++

Le C++ possède ses propres méthodes pour afficher du texte. Elles sont plus faciles à mettre en œuvre que le printf du C.

```
#include <iostream>
...
int a=2;
double pi=3.14;
```

```
std::cout<<pi<<" est superieur a "<<a<<std::endl;
```

4 Recherches personnelles

Les articles courts pourront peut être demander des recherches personnelles sur certains sujets. En particulier, il sera nécessaire d'implanter un détecteur de coins. Voici à titre indicatif, l'url d'un code matlab mettant en œuvre ce détecteur.

http ://www.wisdom.weizmann.ac.il/ deniss/vision_spring04/...
.../files/invariant_features/harris.m

Les articles « longs » sont *a priori* suffisamment explicites pour se suffire à eux mêmes.

5 Recommandations

Certains des sujets proposés ont déjà été donnés les années précédentes. Il est évident que le but de ces projets est la compréhension d'une technique ainsi que la réalisation d'un programme la mettant en oeuvre. La recopie de code source ou de rapport n'entre pas dans ces objectifs, en conséquence je me verrais obligé de sévir de manière tragique si jamais je reconnaissais une source d'inspiration parmi les travaux des années précédentes.

Une attention particulière sera accordée sur la clarté et la pertinence des commentaires et la qualité de la documentation :

De plus, penser à faire un mini-mode d'emploi (un fichier texte d'une demi-page suffit ...) de façon à ce qu'on puisse au moins faire tourner votre programme.

Les sujets sont de difficulté inégale, ceci sera pris en compte lors de la notation.

Avancez régulièrement tant dans la lecture et la compréhension de l'article que dans l'implantation du programme qui en découle. Il me sera difficile de répondre à vos questions la veille au soir de la date limite.

En cas de problème (bloquage, rien compris à l'article, ça compile pas...) n'hésitez pas à me contacter :

- par mail : alexandre.fournier@sophia.inria.fr
- ou par téléphone 04.92.38.77.73

vous pouvez trouver une version électronique de ce document sur cette page :

http ://www-sop.inria.fr/ariana/personnel/Alexandre.Fournier/teaching.php